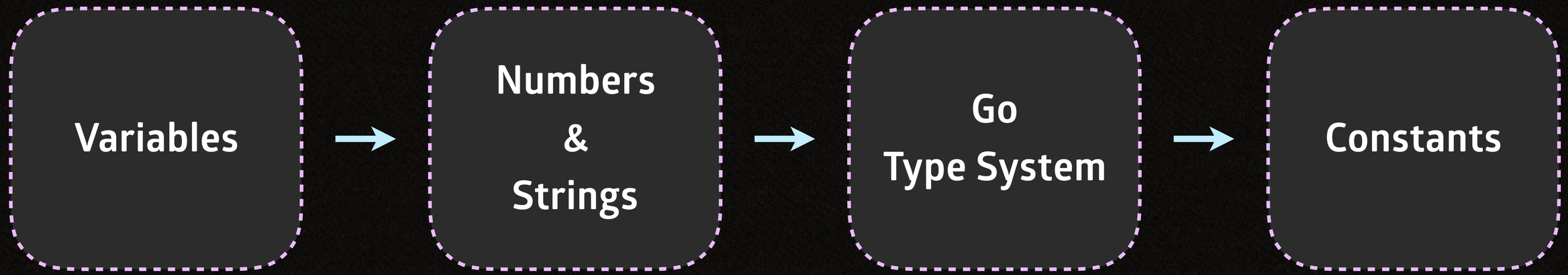
PART II

Fundamentals

Representing Data



VARIABLES

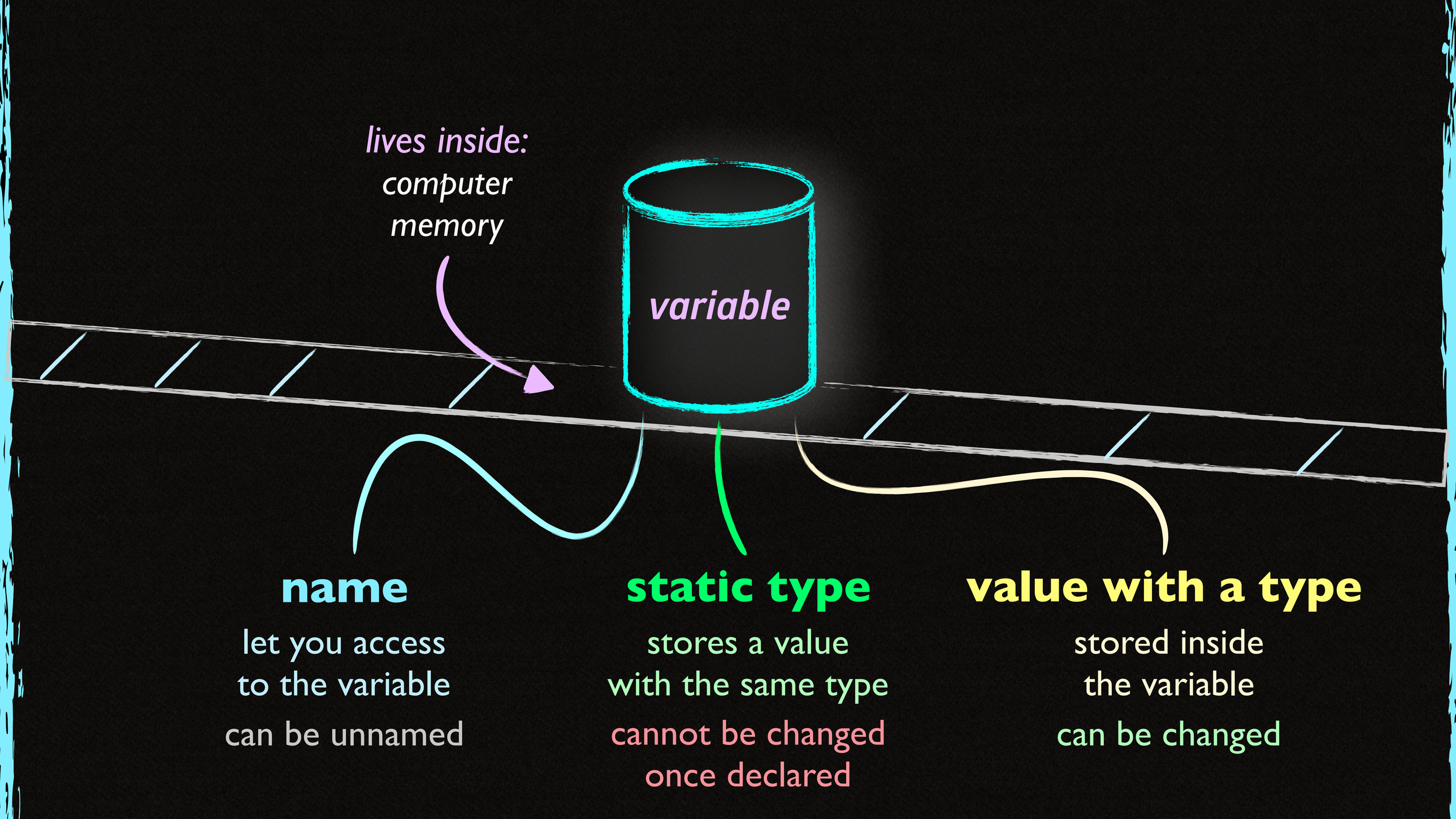
You're going to learn about the basic data types and variables

Basic Data Types → Declarations → Assignment → Type Conversion

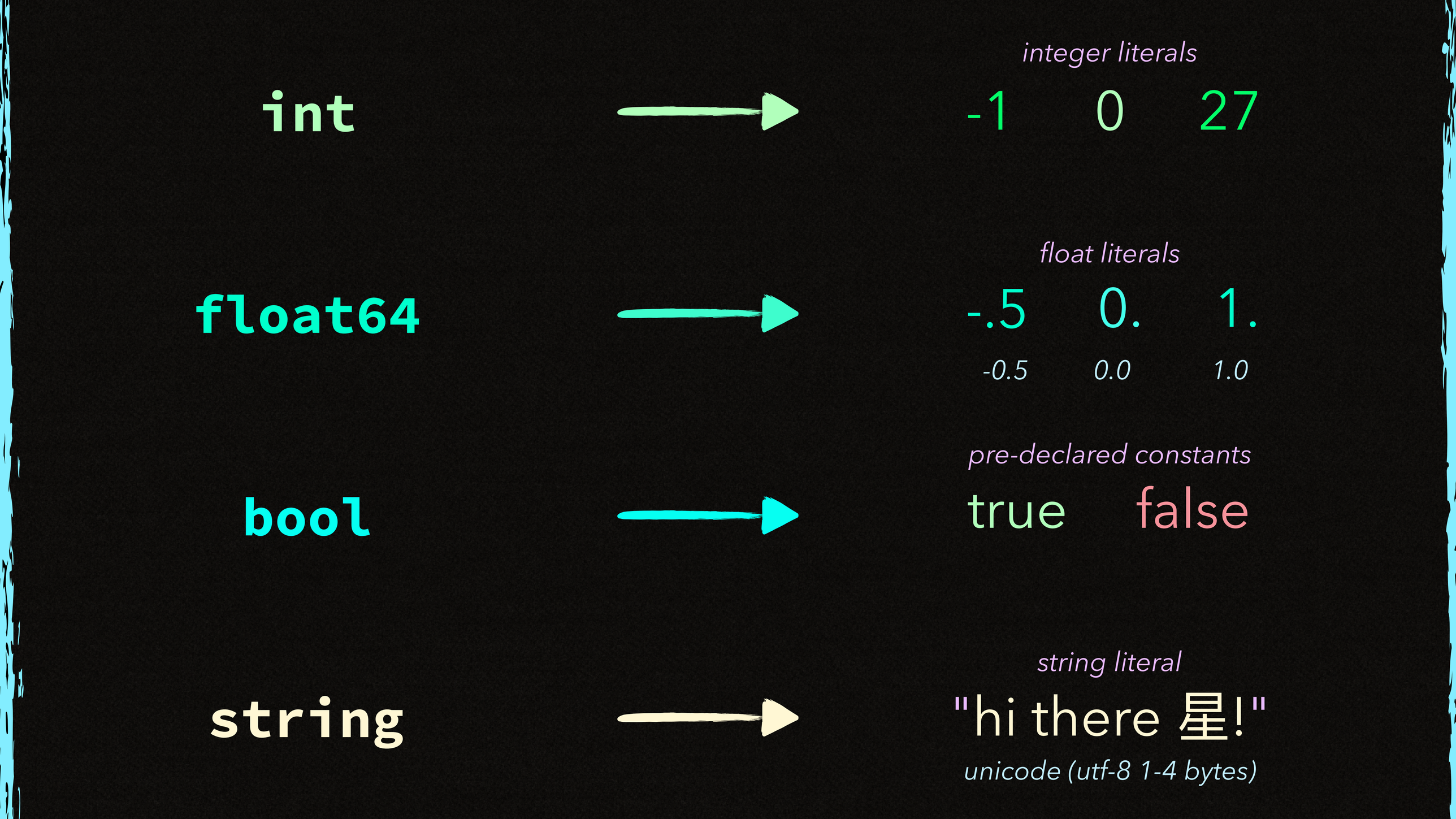
variable declaration

short declaration

redeclaration

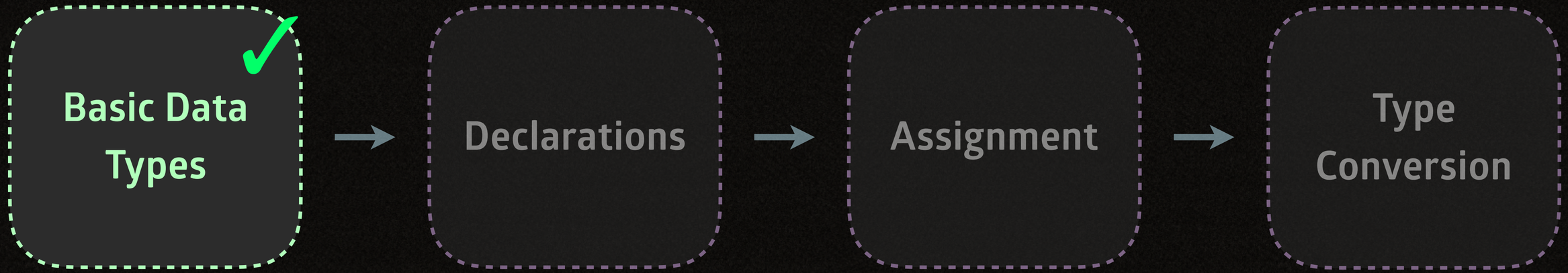

lives inside:
computer
memory
variable
name
let you access
to the variable
can be unnamed
static type
stores a value
with the same type
cannot be changed
once declared
value with a type
stored inside
the variable
can be changed

BASIC DATA TYPES


int
integer literals
-1 0 27
float64
float literals
-.5 0. 1.
-0.5 0.0 1.0
bool
pre-declared constants
true false
string
string literal
"hi there 星!"
unicode (utf-8 1-4 bytes)

VARIABLES

You've completed the basic data types step. Well done!



DECLARATION SYNTAX

you need to declare a variable before you can use it

DECLARATION SYNTAX

Your customer wants you to record various measurements from vehicles

```
package main

func main() {
   var speed int
}
```

DECLARATION SYNTAX

Let's take a look at the parts of the variable declaration

```
var speed int
```

DECLARATION SYNTAX

"var" keyword starts the variable declaration

```
var speed int
```

short for: ***"variable"***

"let's declare a variable"

NAMES

A variable name a value so you can reuse it later

```
var speed int
```

***name** of the variable*

it's also known as an "identifier"

a variable names a value

NAMING RULES

These rules are also true for all names. It's not only about variables.

```
var speed int

var SpeeD int

var _speed int

var 速度 int
```

NAMING RULES

Names should start with a letter or an underscore. Unicode letters are also OK.

```
var 3speed int
```

```
var !speed int
```

```
var spe!ed int
```

```
var var int
```

TYPE

*Variable's type determines what type of values you can store in a variable (**strongly-typed**)*

```
var speed int
```

***static type** of the variable*

*in Go: **every** variable and value has a **type***

DECLARATION SYNTAX

Let's print the variable

```go
package main
import "fmt"

func main() {
   var speed int

   fmt.Println(speed)
}
```

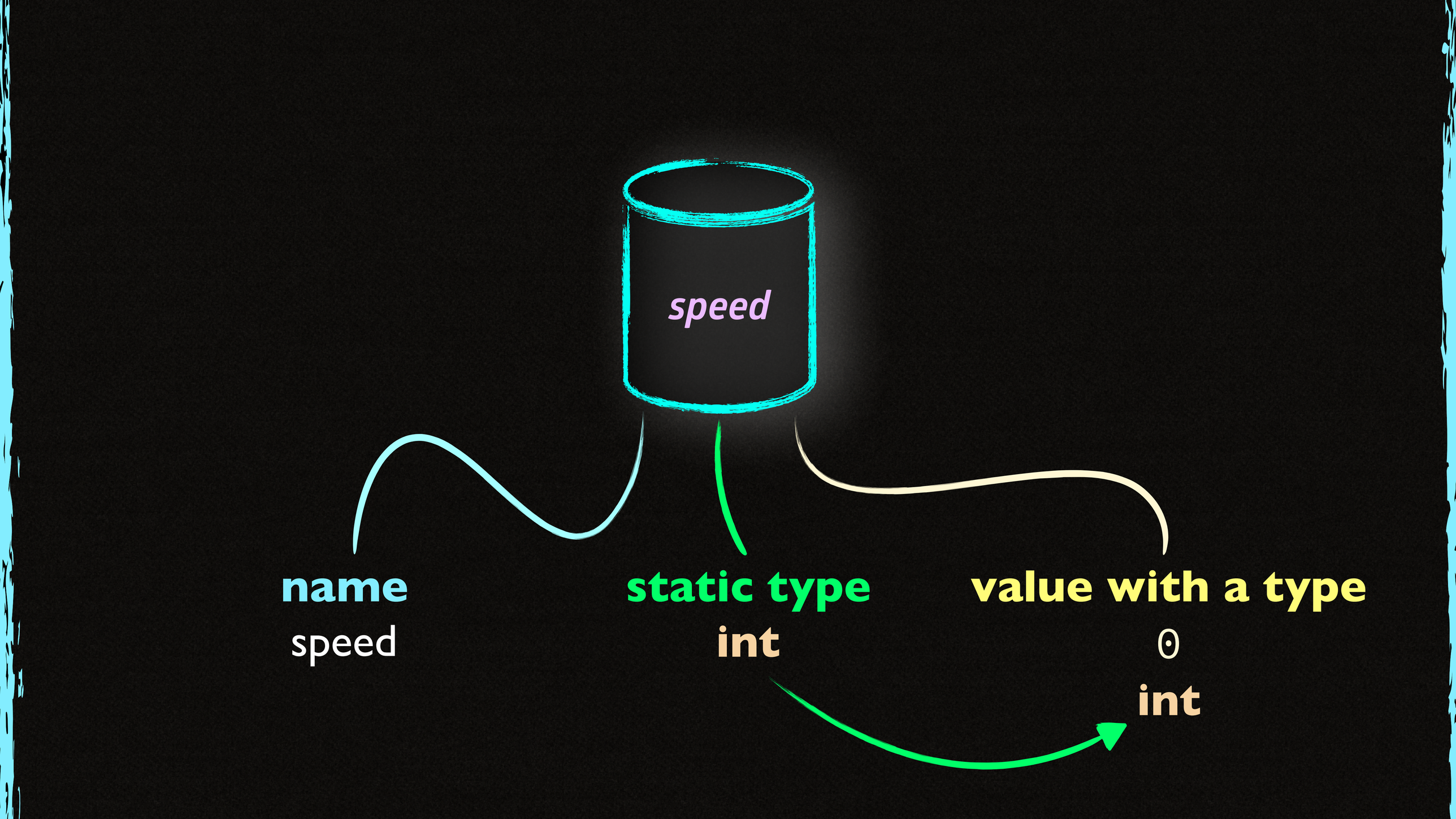
```
$ go run main.go
0
```

ORDER OF STATEMENTS

A variable can only be used after its declaration

```go
package main
import "fmt"

func main() {
   fmt.Println(speed)

   var speed int
}
```

```
$ go run main.go
ERROR: undefined: speed
```


speed
name
speed
static type
int
value with a type
0
int

👉 **This lecture is optional**

EXAMPLE DECLARATIONS

int

... -3 -2 -1 0 1 2 3 ...

```
var nFiles int
var counter int
var nCPU int
```

Integers are mostly used in numerical calculations

float64

... -3.14 -2.1 -1.0 0.0 .1 2.1 2.3 3.14 ...

```
var heat float64
var ratio float64
var degree float64
```

Floats are also used in numerical calculations

bool

true or *false*

```
var off bool
var valid bool
var closed bool
```

Bool data type is mostly used for representing
on / **off** or **valid** / **invalid** states

string

"hello gopher"

```
var msg string
var name string
var text string
```

Mostly used for representing textual data

ZERO VALUES

an empty variable gets a zero value depending on its type

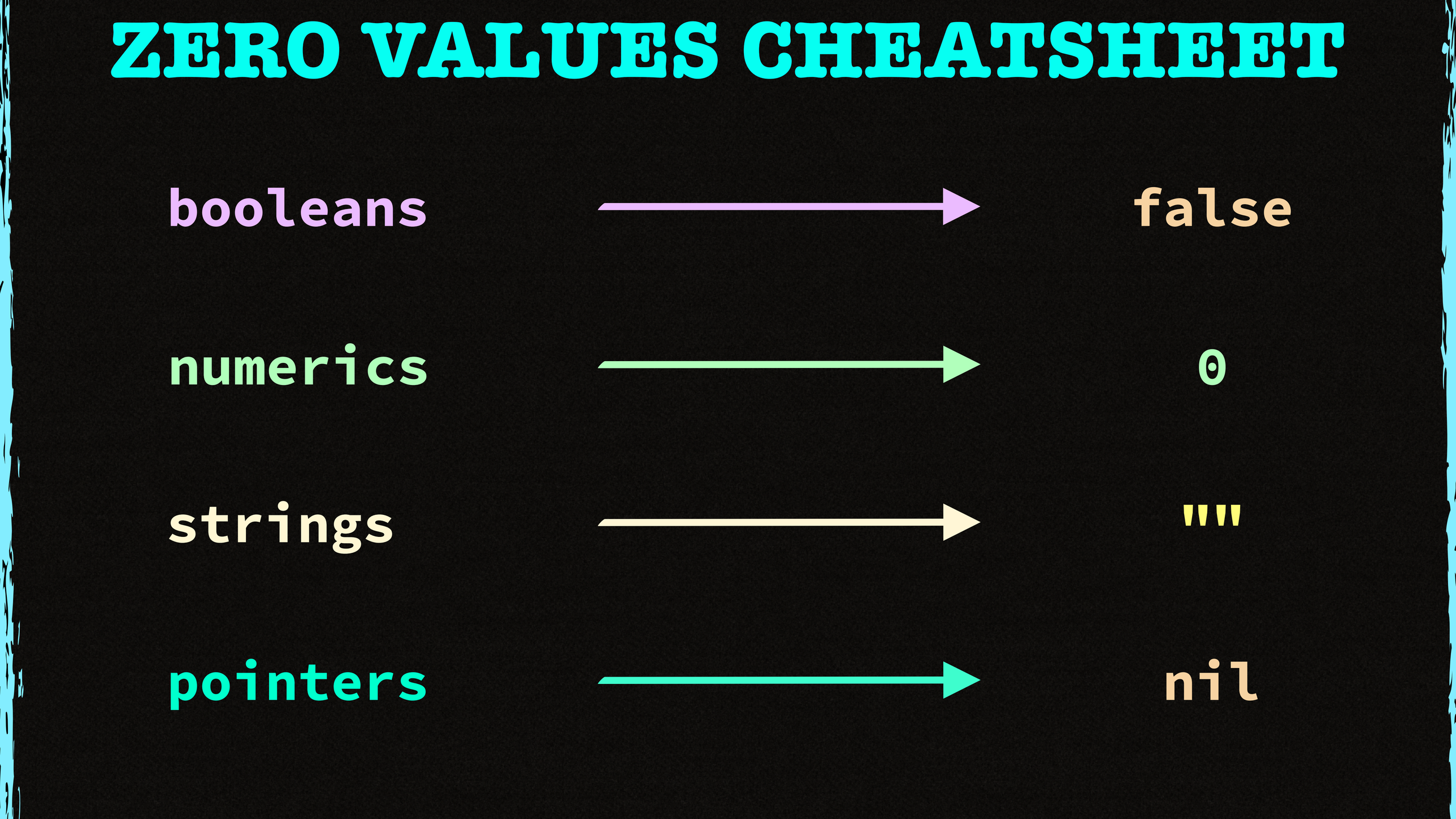
ZERO VALUES

an empty variable gets a zero value depending on its type

```go
package main
import "fmt"

func main() {
   var speed int
   var heat  float64
   var off   bool
   var brand string

   fmt.Println(speed)
   fmt.Println(heat)
   fmt.Println(off)
   fmt.Printf("%q\n", brand)
}
```

```
$ go run main.go
0
0
false
""
```

ZERO VALUES CHEATSHEET



UNUSED VARIABLES

every block-scoped variable should be used

UNUSED VARIABLES

Unused variables cause maintenance nightmares

```go
package main

func main() {
   var speed int
}
```

```
$ go run main.go

ERROR: speed declared and not used
```

UNUSED VARIABLES

Unused variables cause maintenance nightmares

```go
package main
import "fmt"

func main() {
   var speed int

   fmt.Println(speed)
}
```

```
$ go run main.go
0
```

UNUSED VARIABLES

You don't required to use package-level variables

declared at main package's scope

```go
package main
import "fmt"

var speed int

func main() {
}
```

```
$ go run main.go
```

BLANK IDENTIFIER

Swallow the values by using the blank-identifier

```
package main

func main() {
   var speed int

   _ = speed
}
```

blank-identifier

this little guy here is like a ***black-hole****!*

MULTIPLE DECLARATIONS

You can declare multiple variables at once

MULTIPLE DECLARATIONS

You can declare multiple variables at once

```
package main

func main() {
   var speed int
   var heat  float64
   var off   bool
   var brand string

   // ...
}
```

MULTIPLE DECLARATIONS

You can declare multiple variables at once

```
package main

func main() {
   var (
      speed int
      heat  float64
      off   bool
      brand string
   )

   // ...
}
```

MULTIPLE DECLARATIONS

You can declare multiple variables at once

```
package main

func main() {
   var (
      speed int
      heat  float64

      off   bool
      brand string
   )

   // ...
}
```

MULTIPLE DECLARATIONS

You can declare multiple variables using the type once

```go
package main
import "fmt"

func main() {
   var speed, velocity int

   fmt.Println(speed, velocity)
}
```

```go
var speed int
var velocity int
```

INITIALIZATION
&
SHORT DECLARATION

INITIALIZATION

Let's say that you want to declare a variable and you already know its value

```go
package main

func main() {
   var safe bool = true

   fmt.Println(safe)
}
```

```
$ go run main.go
true
```

TYPE INFERENCE

Go can figure out the type automagically

```go
package main

func main() {
  var safe = true

  fmt.Println(safe)
}
```

*Go **deduces** the type of the value automatically*

Then declares the variable using that type

(which is bool here)

```
$ go run main.go
true
```

SHORT DECLARATION

You don't even need to type `var`. Just the name of the variable and its value is enough.

```go
package main

func main() {
    safe := true

    fmt.Println(safe)
}
```

Short Declaration Statement

:=

declares
and
initializes
the variable

```
$ go run main.go
true
```

PACKAGE SCOPE
&
DECLARATION

SHORT DECLARATION

Short declaration works fine in the block scope

```go
package main

func main() {
    safe := true

    fmt.Println(safe)
}
```

SHORT DECLARATION

You can't use short declaration at the package-scope

```go
package main

safe := true

func main() {
   fmt.Println(safe)
}
```

```
$ go run main.go

SYNTAX ERROR:
non-declaration statement
outside function body
```

SHORT DECLARATION

You can't use short declaration at the package-scope

✓

```go
package main

var safe = true

func main() {
   fmt.Println(safe)
}
```

Package-Scoped Variables
Live Forever

Until your program exits

```
$ go run main.go

true
```

SHORT DECLARATION

At the package scope: You can only use declarations that starts with a keyword

They all ***begin*** *with a* ***keyword***

```go
package main

var safe = true

func main() {
    fmt.Println(safe)
}
```

```go
safe := true
```

Short declaration doesn't start with a keyword

So, it cannot be used at the package scope!

MULTIPLE SHORT DECLARATION

MULTIPLE SHORT DECLARATION

You can declare and initialize multiple variables using short declaration

```go
package main

func main() {
    safe, speed := true, 50

    fmt.Println(safe, speed)
}
```

```
$ go run main.go
true 50
```

REDECLARATION

MULTIPLE SHORT DECLARATION

You can both declare and initialize multiple variables using short declaration

```go
package main

func main() {
    safe, speed := true, 50

    fmt.Println(safe, speed)
}
```

```
$ go run main.go
true 50
```

REDECLARATION

Short declaration can initialize new variables and assign to existing variables at the same time

```go
package main

func main() {
    var safe bool
    safe, speed := true, 50

    fmt.Println(safe, speed)
}
```

new variable
declares & initializes
the ***speed***

existing variable
only *assigns* ***true***
to the ***safe***

rule:
at least one
of the variables
should be
a new variable

```
$ go run main.go
true 50
```

SHORT

vs

NORMAL

Use Declaration:

★★★

If you don't know the ***initial value***

★★★

When you need a ***package scoped*** variable

★★★

When you want to ***group variables***
together for greater ***readability***

Use Short Declaration:

★★★

If you know the ***initial value***

★★★

To keep the code ***concise***

★★★

For ***redeclaration***

VARIABLES

You've learned how to declare variables

Basic Data Types ✓ → Declarations ✓ → Assignment → Type Conversion

ASSIGNMENT

changes the value of a variable

ASSIGNMENT

Let's change the value of speed variable using the assignment operator

```go
package main
import "fmt"

func main() {
   var speed int
   fmt.Println(speed)

   speed = 100
   fmt.Println(speed)

   speed = speed - 25
   fmt.Println(speed)
}
```

```
$ go run main.go
0
100
75
```

STRONGLY-TYPED

You can't assign a value with a type to a variable with another type

```
func main() {
    var speed int
    speed = "100"

    var running bool
    running = 1

    var force float64
    speed = force

    force = 1
}
```

✗ `"100"` is **string** but `speed` is **int**

✗ `1` is **number** but `running` is **bool**

✗ `speed` is **int** but `force` is **float64**

✓ `1` is **number** and `force` is **float64**

MULTIPLE ASSIGNMENTS

assign values to multiple variables

MULTIPLE ASSIGNMENTS

You can assign to multiple variables

```go
package main
import ( "fmt"; "time" )

func main() {
  var (
    speed int
    now   time.Time
  )

  speed, now = 100, time.Now()

  fmt.Println(speed, now)
}
```

```
$ go run main.go

100 2130-12-31...
```

easter egg

"Rendezvous with Rama", anyone?

SWAPPING

You can swap values of variables using a multiple assignment

```go
package main
import "fmt"

func main() {
  var (
    speed     = 100
    prevSpeed = 50
  )

  speed, prevSpeed = prevSpeed, speed

  fmt.Println(speed, prevSpeed)
}
```

```
$ go run main.go

50 100
```

PATH SEPARATOR

let's learn how to use multiple result returning expressions
in multiple assignments

PATH SEPARATOR

path.Split()

path package provides utility functions for working with url path strings

PATH SEPARATOR

path.Split()

```
func Split(path string) (dir, file string)
```

PATH SEPARATOR

path.Split()

```
func Split(path string) (dir, file string)
```

dir **string**

file **string**

PATH SEPARATOR

path.Split()

```
func Split(path string) (dir, file string)
```

assets/agreement.pdf

assets/

agreement.pdf

PATH SEPARATOR

You can assign values from a function call to multiple variables

```go
import ( "fmt"; "path" )

func main() {
  var dir, file string

  dir, file = path.Split("css/main.css")

  fmt.Println("dir :", dir)
  fmt.Println("file:", file)
}
```

```
$ go run main.go

dir : css/
file: main.css
```

DISCARDING

You can discard unnecessary values

```
import ( "fmt"; "path" )

func main() {
  var file string

  _, file = path.Split("css/main.css")

  fmt.Println("file:", file)
}
```

```
$ go run main.go

file: main.css
```

SHORT DECLARATION

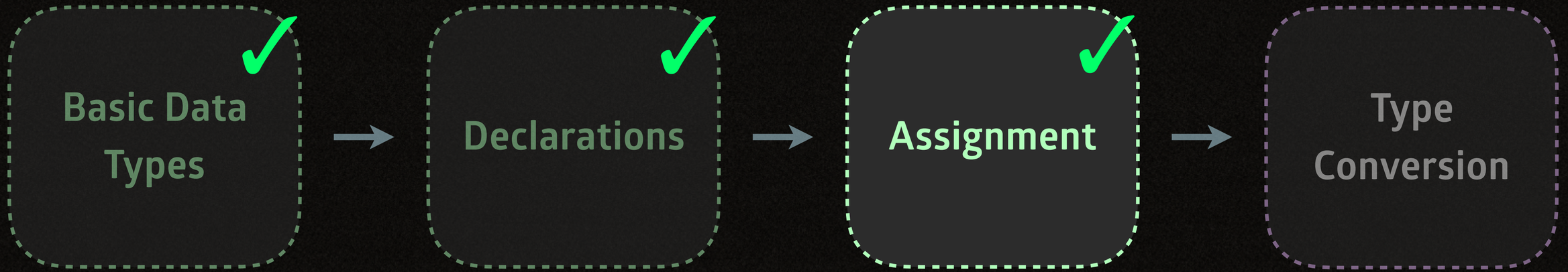
You can also use a short declaration

```go
import ( "fmt"; "path" )

func main() {
  _, file := path.Split("css/main.css")
  fmt.Println("file:", file)
}
```

```
$ go run main.go

file: main.css
```

VARIABLES

You've learned how to assign to variables



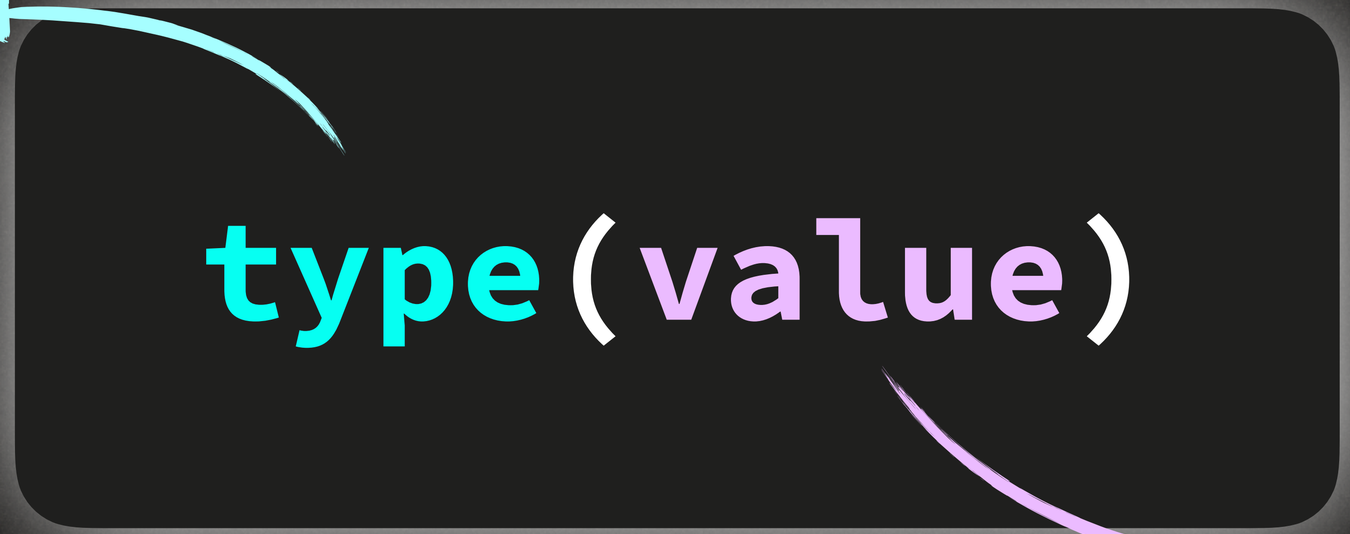
TYPE CONVERSION

changes the type of a value to another type

TYPE CONVERSION

converts a value to another type

name of the type
changes
the type of
a given value
into this type



value
to be converted

TYPE CONVERSION

You can't use values belong to different types together

```
func main() {
   speed  := 100 // speed is int
   force  := 2.5 // force is float64

   speed   = speed * force
}
```

COMPILER ERROR:

*speed * force*

*mismatched types **int** and **float64***

TYPE CONVERSION

You need to explicitly convert the values

```go
func main() {
  speed := 100 // speed is int
  force := 2.5 // force is float64

  speed = speed * int(force)

  fmt.Println(speed)
  fmt.Println(force, int(force))
}
```

2.5 becomes 2

```
$ go run main.go
200
2.5 2
```

force variable's value is still 2.5

conversion can be a destructive operation

TYPE CONVERSION

Which value you convert first matters

```go
func main() {
    speed := 100   // speed is int
    force := 2.5   // force is float64

    speed = int( float64(speed) * force )

    fmt.Println(speed)
}
```

```
$ go run main.go
250
```

`float64(100)` = 100.0 → float64

100.0 * 2.5 = 250.0 → float64

`int(250.0)` = 250 → int

TYPE CONVERSION #2

types with different names are different types

TYPE CONVERSION

int and int32 are different types

```
func main() {
    var apple  int
    var orange int32

    apple = orange
}
```

COMPILER ERROR:
cannot use ***orange*** *(type* ***int32****)*
as type **int**
in assignment

TYPE CONVERSION

types with different names are different types

```
func main() {
   var apple  int
   var orange int32

   apple = int(orange)

   fmt.Println(apple)
}
```

✓

int(orange) = int

apple = int

```
$ go run main.go
0
```

GREETER!

get input from the command line

+ intro to slices

OS PACKAGE

os = operating system

allows you to access to
operating system functionalities

INTRO TO SLICES

Args *variable belongs to the* **os package**

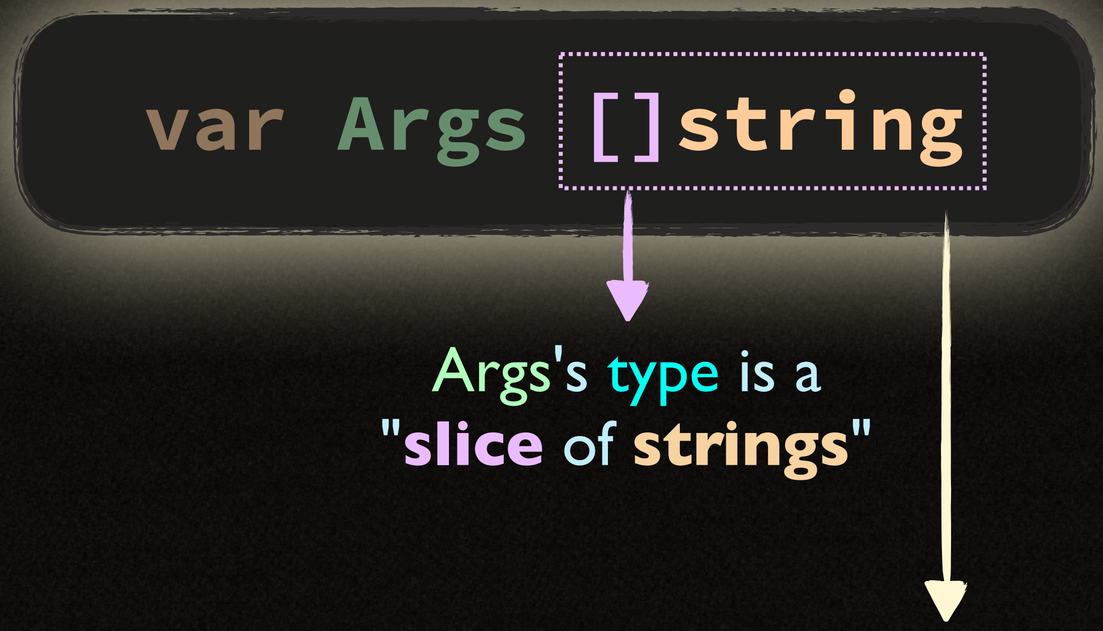
```
var Args []string
```

When you run a Go program

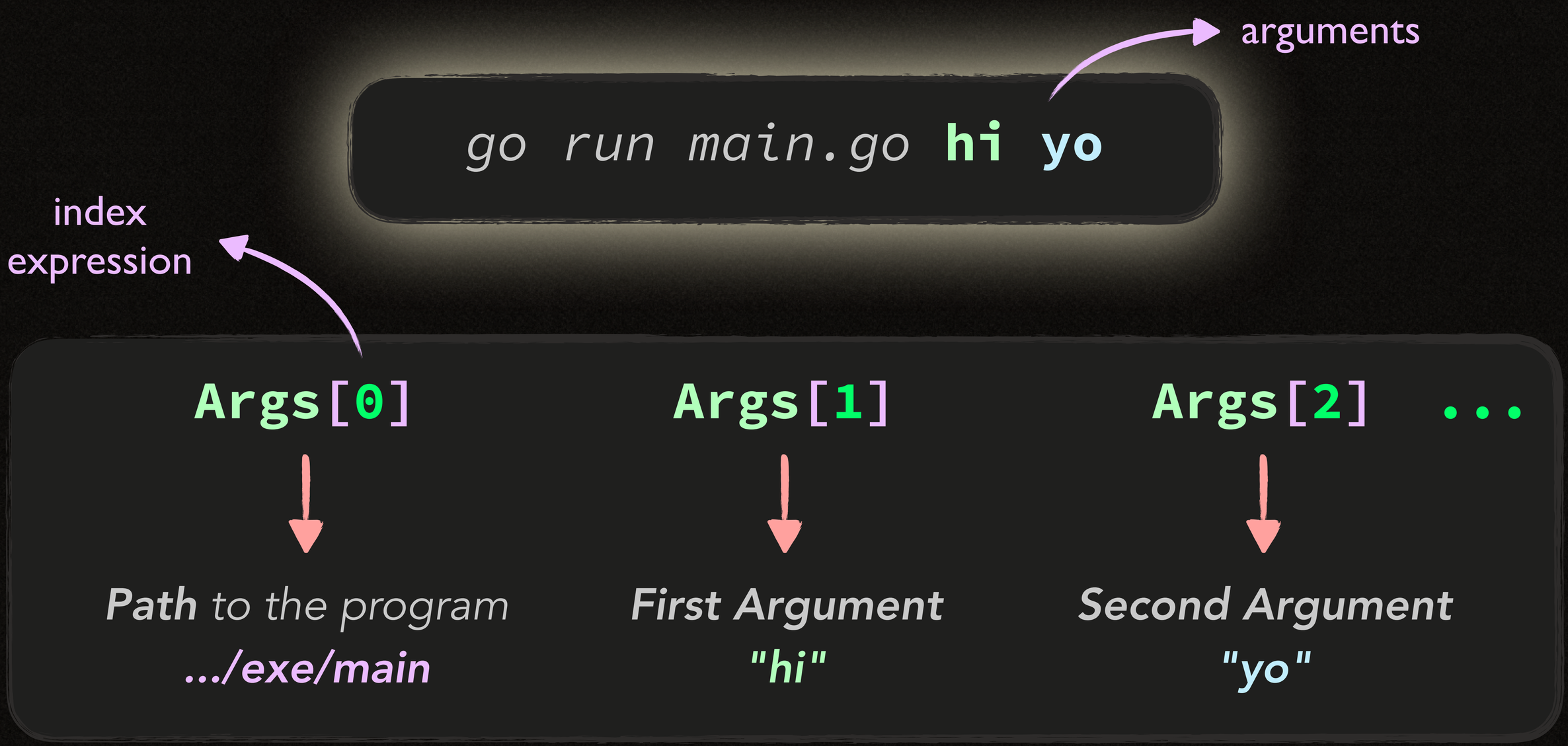
Go puts the command-line arguments into this variable automatically

INTRO TO SLICES

A slice can store multiple values

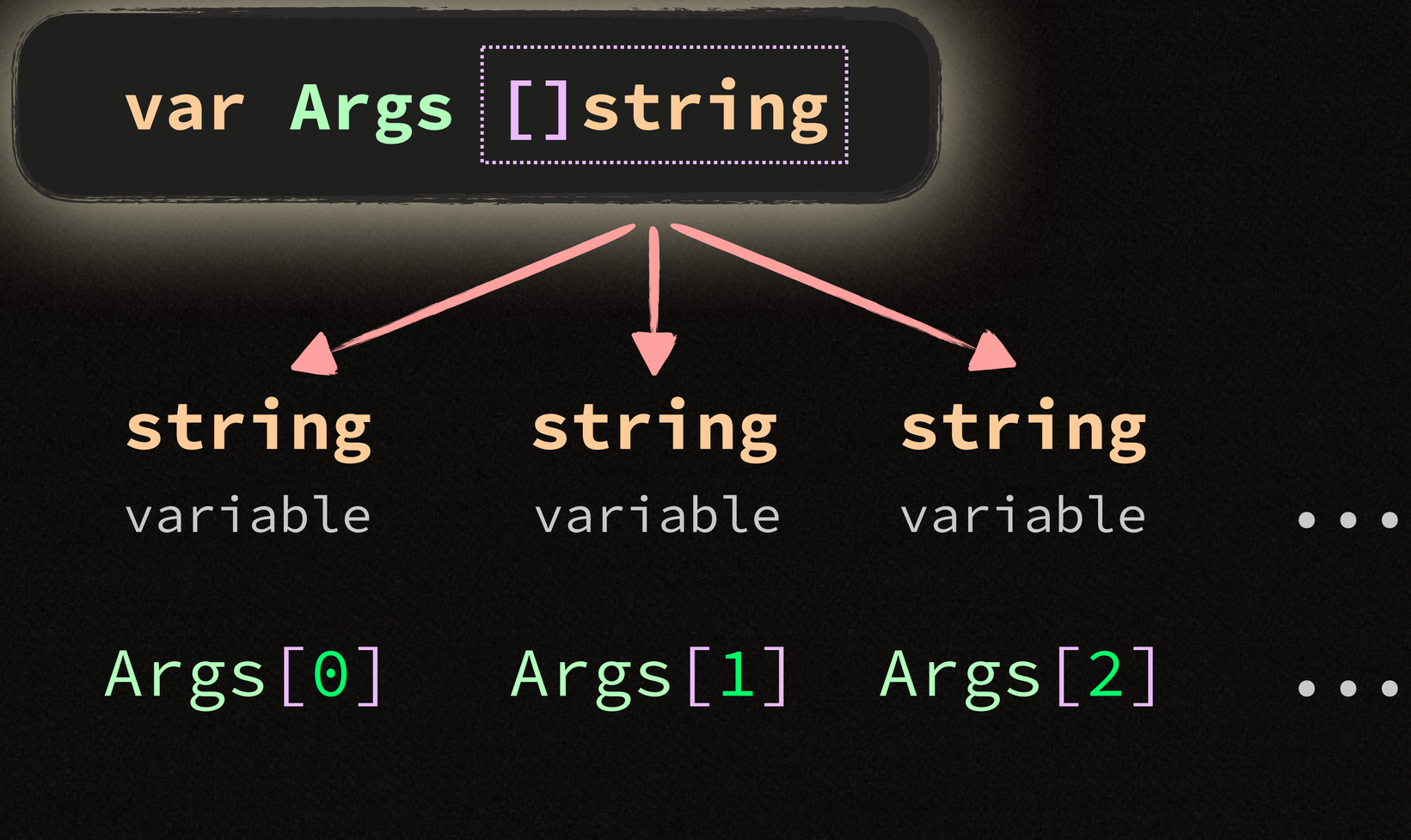

var Args []string
Args's type is a
"slice of strings"
Args slice can store multiple string values
each value inside slice is an unnamed variable

INDEX EXRESSION



INTRO TO SLICES

a slice stores unnamed variables inside



GREETER!

get input from the command line

+ intro to slices

GREETER!

get input from the command line

+ intro to slices

EXERCISE

greet more people

1. Greet more people
2. Change Gandalf

go run main.go inanc lina ebru

❤️

summary

declaring variables

```
var long string
```

variable declaration

```
short := "short"
```

short declaration

go is statically-typed

```
var name string
```

static type

it cannot be changed after the declaration

zero values

"make the zero-value useful"

— *Rob Pike*

declared variables should be used

```
stage := "init"
    _ = stage
```

initializing variables

```
var long string = "init"
```

initializer value

initializer expression

type inference

```
var long = "init"
```

Go **infers** that "init" is a string

Then it declares the long variable's type as a string

redeclaration

```
name := "Nikola"
```

Assignment & Declaration:

```
name = "Grace"
age := 86
```

Redeclaration:

```
name, age := "Grace", 86
```

assigns to the name
declares the age

assigning to variables

variable = expression

variable, variable2 = expression, expression2

assignment using expressions

```
variable = value + value2

variable = function()

variable, variable2 = anotherFunction()
```

swapping

```
variable, variable2 = variable2, variable
```

discarding using blank-identifier

```
variable, _ = value, value2
```

```
variable, _ := value, value2
```

hulk-level strongly-typed

```
var ratio float64
var speed int

speed = ratio
```

type conversion

```
var ratio float64
var speed int

speed = int(ratio)
```

✓

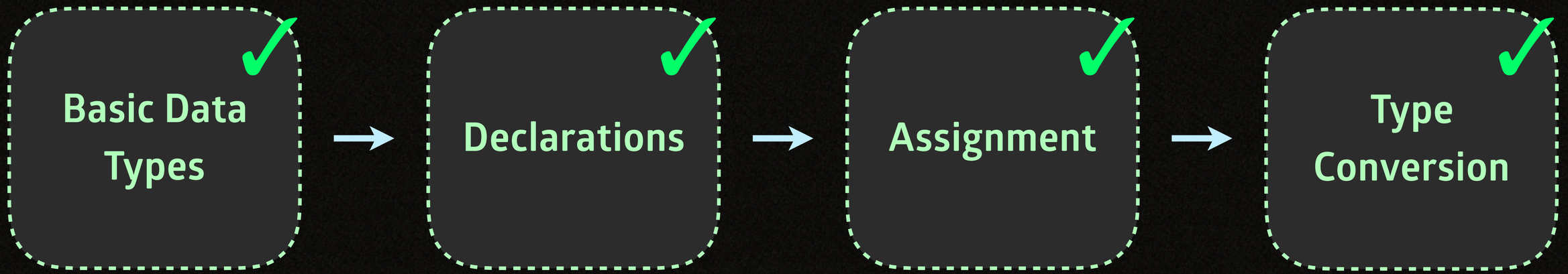
types with different names are different types

path.Split()

os.Args stores the command-line arguments

```
var Args []string
```

```
Args[0]
Args[1]
...
```

VARIABLES

You've completed all the steps! Congrats!